Universitat de Girona
Escola Politècnica Superior

Pràctiques ESII

Curs 2023/24

GEINF- GDDV

Pràctica 4

Grup T
Jordi Badia, Aniol Juanola i Guillem Vidal

(December 7, 2023)

Contents

1 *Decorator* analysis **2**

2 Class diagram **4**

3 *Decorator* example **4**

3.1 Problem description 4

3.2 Class diagram 4

1 *Decorator* analysis

The Decorator pattern allows us to add behaviour to an object dynamically, without the need of specifying all the combinations at compile time. This is particularly useful when dealing with effects on the vehicles in the race, since they are added dynamically to each vehicle, and the same effect can be applied to multiple at a time. In order to achieve this, we separated the concept of vehicle into two: the motor and the vehicle itself. The motor receives the current state of the vehicle (velocity, angle, etc) and the player input and computes how would each variable change in the next frame.[1] As such, the class `Vehicle` and the interface `IMotor` form an strategy pattern, where the `Vehicle` does not have to know which `IMotor` is using to move. `IMotor` can then be decorated to add the different effects of the race, like, for example, slow downs or slippery terrain (the first one would reduce the increment of speed, the second the change in angle).

Adding behaviour to an object is really simple using the Decorator, it becomes problematic whenever one want to remove behaviours. Since `Vehicle` does not know what `IMotor` has, even less whether it is decorated, it is impossible for Vehicle, imperative in its design, to know how is `IMotor` structured. To solve this problem, `MotorDecorator` is a doubly-linked list.

Each `MotorDecorator` object knows for sure one thing: its wrapper and wrappee[2] are never null. This is because the last wrappee is always going to be an instance of `IMotor` (Quadriga, Biga, etc) and the first wrapper is always going to be `Vehicle`. This simplifies the development of deleting behaviours, because there are no particular cases of the nodes being null. However, this means that the previous node (the wrapper) is an attribute that can contain either a `MotorDecorator` or a `Vehicle`, because of this, we need an interface defining a motor wrapper, which has only one method: setting its wrappee (the node it points to). This method allows us to remove the `MotorDecorator` we want by calling `setWrappee` on its previous node and passing as parameter the next, removing the current `MotorDecorator` from the list itself. The only criticism of this implementation is that there necessarily is a instanceof[3] of `IMotor` to `MotorDecorator`, see Listing 1.

[1] It returns change, that means that vehicle is then responsible of adding such change to the variables in its state.

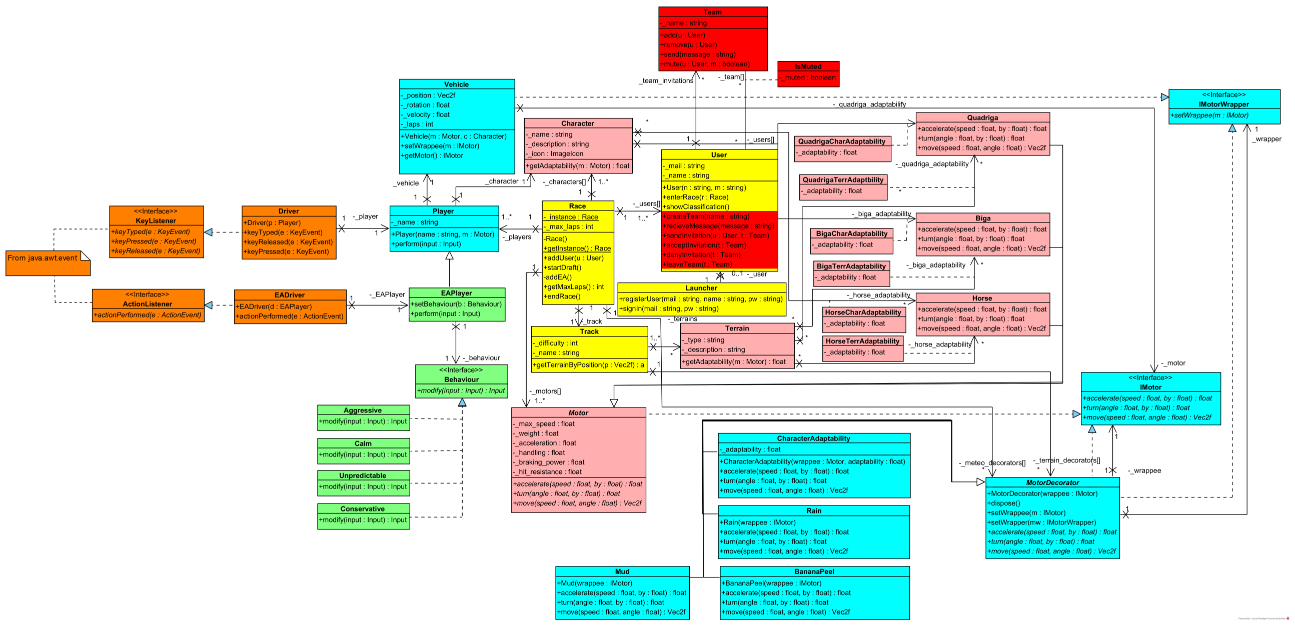
[2] Wrapper defines the previous node in the list and wrappee the next one.

[3] It could be resolved by making a `IMotorWrappee` interface that is implemented by `IMotor` and `MotorDecorator`. However, we do not know how much it would break SOLID, given that `IMotor` would implement a functionality relating its decorator.

```java
interface IMotor {}

interface IMotorWrapper {

    void setWrappee(IMotor motor);

}

class Vehicle {

    private IMotor motor;

    public void setWrappee(IMotor motor) {

        this.motor = motor;

        if (motor instanceof MotorDecorator) {

            ((MotorDecorator)motor).setWrapper(this);

        }

    }

}

abstract class MotorDecorator implements IMotor, IMotorWrapper {

    private IMotorWrapper wrapper;
    private IMotor wrappee;

    public MotorDecorator(IMotor wrappee) {

        this.wrapper = null;
        this.wrappee = wrappee;

    }

    public void setWrappee(IMotor motor) { /* Same implementation as Vehicle. */ }

    public void setWrapper(IMotorWrapper wrapper) {

        this.wrapper = wrapper;

    }

}
```

Listing 1: Implementation example of the Decorator pattern in motor.

2 Class diagram



3 *Decorator* example

3.1 Problem description

The proposed situation consists of an HTML text formatter. The Decorator::main() function contains a `TextFormatter` (which real type is `HTMLTextFormatter`) and allows to build the decorated formatter.

When a value is read and it isn't a menu option, `TextFormatter::getHTML(string)` is called, triggering de consequent chain reaction of calls.

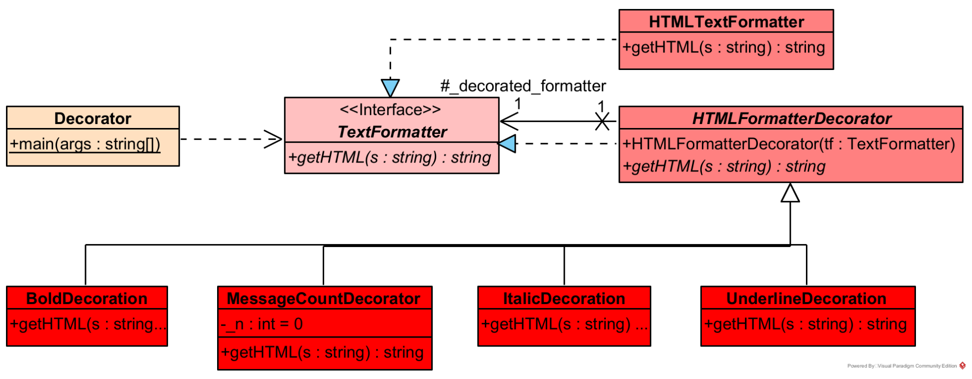
`HTMLTextFormatter` would be the core of the "onion", and it will be contained by all the decorators that the `Decorator::main()` has added.

Decorators will make calls like:

```
return "<b>" + _decorated_formatter.getHTML(s) + "</b>";
```

`MessageCountDecorator` adds an index before each message, creating a numbered list. That's why it has a private attribute that is initialized to 0.

3.2 Class diagram



HTMLTextFormatter
+getHTML(s : string) : string
#_decorated_formatter
1
1
Decorator
+main(args : string[])
<<Interface>>
TextFormatter
+getHTML(s : string) : string
HTMLFormatterDecorator
+HTMLFormatterDecorator(tf : TextFormatter)
+getHTML(s : string) : string
BoldDecoration
+getHTML(s : string...
MessageCountDecorator
-_n : int = 0
+getHTML(s : string) : string
ItalicDecoration
+getHTML(s : string) ...
UnderlineDecoration
+getHTML(s : string) : string
Powered By: Visual Paradigm Community Edition